Node.js Stream - 进阶篇

本篇介绍 stream 是如何实现流式数据处理的。

数据生产和消耗的媒介

为什么使用流取数据

下面是一个读取文件内容的例子：

```
const fs = require('fs')fs.readFile(file, function (err, body) {
  console.log(body)
  console.log(body.toString())
})
```

但如果文件内容较大，譬如在 440M 时，执行上述代码的输出为：

```
<Buffer 64 74 09 75 61 09 63 6f 75 6e 74 0a 0a 64 74 09 75 61 09 63 6f 75 6e 74 0a 32 30 31 35 31 32 30 38 09 4d 6f 7a 69 6c 6c 61 2f 35 2e 30 20 28 63 6f 6d ... >

buffer.js:382
    throw new Error('toString failed');
    ^

Error: toString failed
    at Buffer.toString (buffer.js:382:11)
```

报错的原因是 body 这个 Buffer 对象的长度过大，导致 toString 方法失败。 可见，这种一次获取全部内容的做法，不适合操作大文件。

可以考虑使用流来读取文件内容。

```
const fs = require('fs')fs.createReadStream(file).pipe(process.stdout)
```

fs.createReadStream 创建一个可读流，连接了源头（上游，文件）和消耗方（下游，标准输出）。

执行上面代码时，流会逐次调用 fs.read，将文件中的内容分批取出传给下游。在文件看来，它的内容被分块地连续取走了。 在下游看来，它收到的是一个先后到达的数据序列。 如果不需要一次操作全部内容，它可以处理完一个数据便丢掉。 在流看来，任一时刻它都只存储了文件中的一部分数据，只是内容在变化而已。

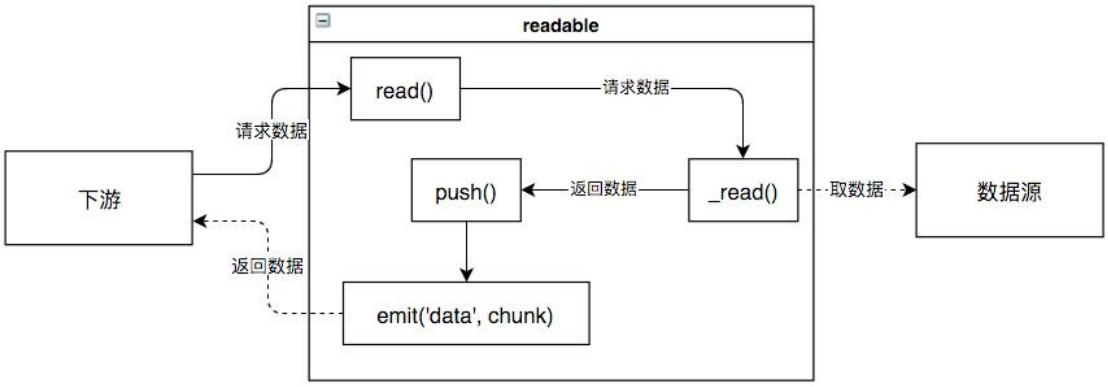
这种情况就像是用水管去取池子中的水。 每当用掉一点水，水管便会从池子中再取出一点。 无论水池有多大，都只存储了与水管容积等量的水。

如何通过流取到数据

用 Readable 创建对象 readable 后，便得到了一个可读流。 如果实现_read 方法，就将流连接到一个底层数据源。 流通过调用_read 向底层请求数据，底层再调用流的 push 方法将需要的数据传递过来。

当 readable 连接了数据源后，下游便可以调用 readable.read(n)向流请求数据，同时监听 readable 的 data 事件来接收取到的数据。

这个流程可简述为：



read

read 方法中的逻辑可用下图表示，后面几节将对该图中各环节加以说明。

push 方法

消耗方调用 read(n)促使流输出数据，而流通过_read()使底层调用 push 方法将数据传给流。

如果流在流动模式下（state.flowing 为 true）输出数据，数据会自发地通过 data 事件输出，不需要消耗方反复调用 read(n)。

如果调用 push 方法时缓存为空，则当前数据即为下一个需要的数据。 这个数据可能先添加到缓存中，也可能直接输出。 执行 read 方法时，在调用_read 后，如果从缓存中取到了数据，就以 data 事件输出。

所以，如果_read 异步调用 push 时发现缓存为空，则意味着当前数据是下一个需要的数据，且不会被 read 方法输出，应当在 push 方法中立即以 data 事件输出。

因此，上图中"立即输出"的条件是：

```
state.flowing && state.length === 0 && !state.sync
```

end 事件

由于流是分次向底层请求数据的，需要底层显示地告诉流数据是否取完。 所以，当某次（执行_read()）取数据时，调用了 push(null)，就意味着底层数据取完。此时，流会设置 state.ended。

state.length 表示缓存中当前的数据量。只有当 state.length 为 0，且 state.ended 为 true，才意味着所有的数据都被消耗了。 一旦在执行 read(n)时检测到这个条件，便会触发 end 事件。 当然，这个事件只会触发一次。

readable 事件

在调用完_read()后，read(n)会试着从缓存中取数据。 如果_read()是异步调用 push 方法的，则此时缓存中的数据量不会增多，容易出现数据量不够的现象。

如果 read(n)的返回值为 null，说明这次未能从缓存中取出所需量的数据。此时，消耗方需要等待新的数据到达后再次尝试调用 read 方法。

在数据到达后，流是通过 readable 事件来通知消耗方的。 在此种情况下，push 方法如果立即输出数据，接收方直接监听 data 事件即可，否则数据被添加到缓存中，需要触发 readable 事件。 消耗方必须监听这个事件，再调用 read 方法取得数据。

doRead

流中维护了一个缓存，当缓存中的数据足够多时，调用 read()不会引起_read()的调用，即不需要向底层请求数据。 用 doRead 来表示 read(n)是否需要向底层取数据，其逻辑为：

```
var doRead = state.needReadable

if (state.length === 0 || state.length - n < state.highWaterMark) {

  doRead = true

}

if (state.ended || state.reading) {

  doRead = false

}

if (doRead) {

  state.reading = true

  state.sync = true

  if (state.length === 0) {

    state.needReadable = true

  }

  this._read(state.highWaterMark)

  state.sync = false

}
```

state.reading 标志上次从底层取数据的操作是否已完成。 一旦 push 方法被调用，就会设置为 false，表示此次_read()结束。

state.highWaterMark 是给缓存大小设置的一个上限阈值。 如果取走 n 个数据后，缓存中保有的数据不足这个量，便会从底层取一次数据。

howMuchToRead

调用 read(n)去取 n 个数据时， m = howMuchToRead(n)是将从缓存中实际获取的数据量。 根据以下几种情况赋值，一旦确定则立即返回：

- state.length 为 0，state.ended 为 true。 数据源已枯竭，且缓存为空，无数据可取，m 为 0.
- state.objectMode 为 true。 n 为 0，则 m 为 0； 否则 m 为 1，将缓存的第一个元素输出。
- n 是数字。 若 n <= 0，则 m 为 0； 若 n > state.length，表示缓存中数据量不够。 此时如果还有数据可读（state.ended 为 false），则 m 为 0，同时设置 state.needReadable，下次执行 read()时 doRead 会为 true，将从底层再取数据。 如果已无数据可读（state.ended 为 true），则 m 为 state.length，将剩下的数据全部输出。 若 0 < n <= state.length，则缓存中数据够用，m 为 n。
- 其它情况。 state.flowing 为 true（流动模式），则 m 为缓存中第一个元素(Buffer)的长度,实则还是将第一个元素输出；否则 m 为 state.length，将缓存读空。

上面的规则中：

- n 通常是 undefined 或 0，即不指定读取的字节数。
- read(0)不会有数据输出，但从前面对 doRead 的分析可以看出，是有可能从底层读取数据的。
- 执行 read()时，由于流动模式下数据会不断输出，所以每次只输出缓存中第一个元素输出，而非流动模式则会将缓存读空。
- objectMode 为 true 时，m 为 0 或 1。此时，一次 push()对应一次 data 事件。

总结

可读流是获取底层数据的工具，消耗方通过调用 read 方法向流请求数据，流再从缓存中将数据返回,或以 data 事件输出。如果缓存中数据不够,便会调用_read 方法去底层取数据。 该方法在拿到底层数据后，调用 push 方法将数据交由流处理（立即输出或存入缓存）。

可以结合 readable 事件和 read 方法来将数据全部消耗，这是暂停模式的消耗方法。 但更常见的是在流动模式下消耗数据，具体见后面的章节。

数据的流式消耗

所谓"流式数据"，是指按时间先后到达的数据序列。

数据消耗模式

可以在两种模式下消耗可读流中的数据：暂停模式（paused mode）和流动模式（flowing mode）。

流动模式下，数据会源源不断地生产出来，形成"流动"现象。 监听流的 data 事件便可进入该模式。

暂停模式下，需要显示地调用 read()，触发 data 事件。

可读流对象 readable 中有一个维护状态的对象，readable._readableState，这里简称为 state。 其中有一个标记，state.flowing， 可用来判别流的模式。 它有三种可能值：

- true。流动模式。
- false。暂停模式。
- null。初始状态。

调用 readable.resume()可使流进入流动模式，state.flowing 被设为 true。 调用 readable.pause()可使流进入暂停模式， state.flowing 被设为 false。

暂停模式

在初始状态下，监听 data 事件，会使流进入流动模式。 但如果在暂停模式下，监听 data 事件并不会使它进入流动模式。 为了消耗流，需要显示调用 read()方法。

```
const Readable = require('stream').Readable

// 底层数据 const dataSource = ['a', 'b', 'c']

const readable = Readable()readable._read = function () {

  if (dataSource.length) {
```

```
    this.push(dataSource.shift())
  } else {
    this.push(null)
  }
}
// 进入暂停模式 readable.pause()readable.on('data', data =>
process.stdout.write('\ndata: ' + data))
var data = readable.read()while (data !== null) {
  process.stdout.write('\nread: ' + data)
  data = readable.read()
}
```

执行上面的脚本，输出如下：

```
data: a
read: a
data: b
read: b
data: c
read: c
```

可见，在暂停模式下，调用一次 read 方法便读取一次数据。 执行 read()时，如果缓存中数据不够，会调用_read()去底层取。_read 方法中可以同步或异步地调用 push(data)来将底层数据交给流处理。

在上面的例子中，由于是同步调用 push 方法，数据会添加到缓存中。 read 方法在执行完_read 方法后，便从缓存中取数据，再返回，且以 data 事件输出。

如果改成异步调用 push 方法，则由于_read()执行完后，数据来不及放入缓存，将出现 read()返回 null 的现象。 见下面的示例：

```
const Readable = require('stream').Readable

// 底层数据 const dataSource = ['a', 'b', 'c']

const readable = Readable()readable._read = function () {

  process.nextTick(() => {

    if (dataSource.length) {

      this.push(dataSource.shift())

    } else {

      this.push(null)

    }

  })
}

readable.pause()readable.on('data', data => process.stdout.write('\ndata: ' +
data))

while (null !== readable.read()) ;
```


执行上述脚本，可以发现没有任何数据输出。

此时，需要使用 readable 事件：

```
const Readable = require('stream').Readable

// 底层数据 const dataSource = ['a', 'b', 'c']

const readable = Readable()readable._read = function () {

  process.nextTick(() => {

    if (dataSource.length) {

      this.push(dataSource.shift())

    } else {

      this.push(null)
```

```
    }
  })
}
readable.pause()readable.on('data', data => process.stdout.write('\ndata: ' +
data))
readable.on('readable', function () {
  while (null !== readable.read()) ;;
})
```

输出：

```
data: a
data: b
data: c
```

当 read()返回 null 时，意味着当前缓存数据不够，而且底层数据还没加进来（异步调用 push()）。 此种情况下 state.needReadable 会被设置为 true。 push 方法被调用时，由于是暂停模式，不会立即输出数据，而是将数据放入缓存，并触发一次 readable 事件。

所以，一旦 read 被调用，上面的例子中就会形成一个循环：readable 事件导致 read 方法调用，read 方法又触发 readable 事件。

首次监听 readable 事件时，还会触发一次 read(0)的调用，从而引起_read 和 push 方法的调用，从而启动循环。

总之，在暂停模式下需要使用 readable 事件和 read 方法来消耗流。

流动模式

流动模式使用起来更简单一些。

一般创建流后，监听 data 事件，或者通过 pipe 方法将数据导向另一个可写流，即可进入流动模式开始消耗数据。 尤其是 pipe 方法中还提供了背压机制，所以使用 pipe 进入流动模式的情况非常普遍。

本节解释 data 事件如何能触发流动模式。

先看一下 Readable 是如何处理 data 事件的监听的：

```
Readable.prototype.on = function (ev, fn) {

  var res = Stream.prototype.on.call(this, ev, fn)

  if (ev === 'data' && false !== this._readableState.flowing) {

    this.resume()

  }


  // 处理 readable 事件的监听

  // 省略


  return res

}
```


Stream 继承自 EventEmitter，且是 Readable 的父类。 从上面的逻辑可以看出，在将 fn 加入事件队列后，如果发现处于非暂停模式，则会调用 this.resume()，开始流动模式。

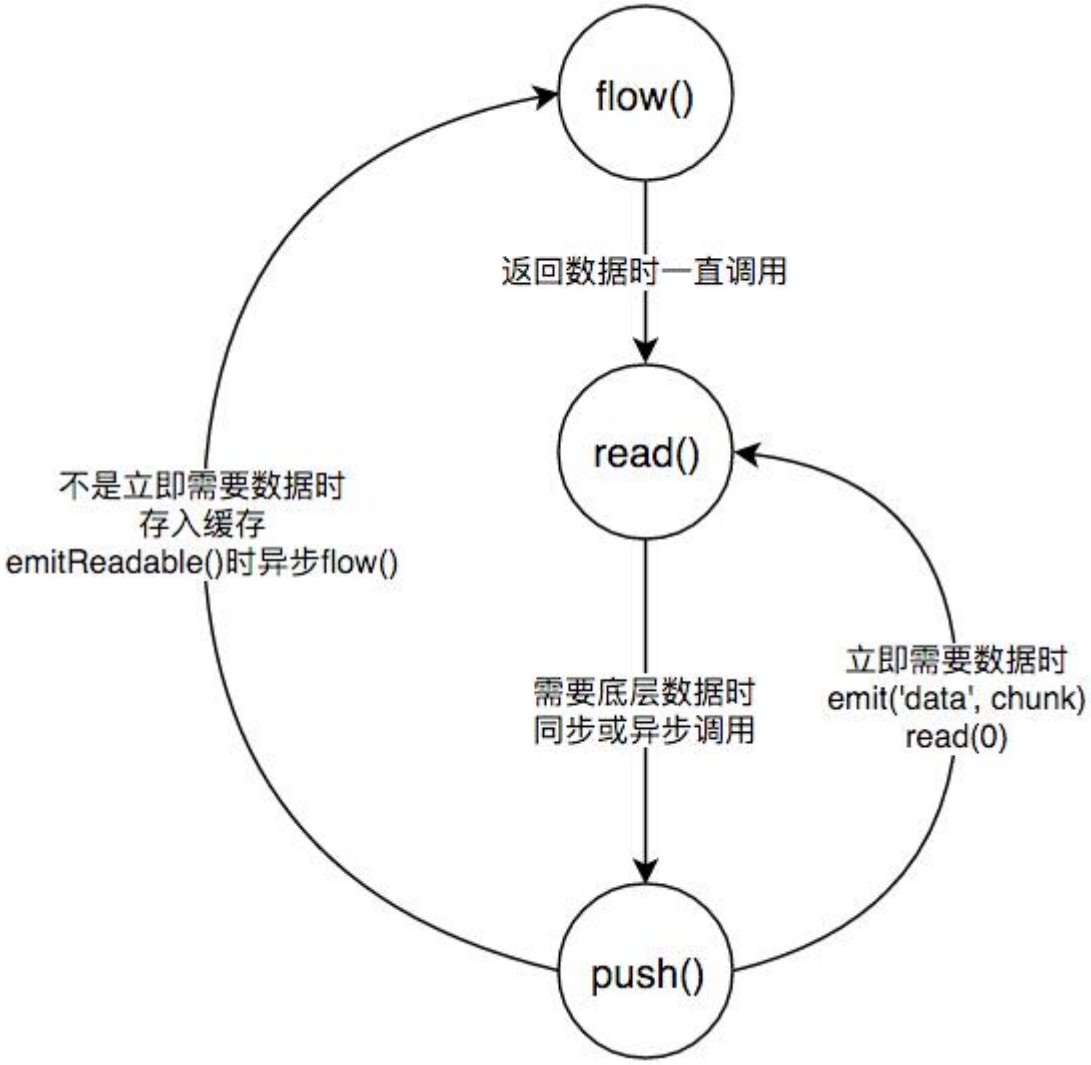
resume()方法先将 state.flowing 设为 true， 然后会在下一个 tick 中执行 flow，试图将缓存读空：

```
if (state.flowing) do {

  var chunk = stream.read()

} while (null !== chunk && state.flowing)
```

flow 中每次 read()都可能触发 push()的调用， 而 push()中又可能触发 flow()或 read()的调用， 这样就形成了数据生生不息的流动。 其关系可简述为：



下面再详细看一下 push()的两个分支：

```
if (state.flowing && state.length === 0 && !state.sync) {
  stream.emit('data', chunk)
  stream.read(0)
} else {
  state.length += state.objectMode ? 1 : chunk.length
  state.buffer.push(chunk)

  if (state.needReadable)
    emitReadable(stream)
```

```
}
```

称第一个分支为立即输出。

在立即输出的情况下，输出数据后，执行 read(0)，进一步引起_read()和 push()的调用，从而使数据源源不断地输出。

在非立即输出的情况下，数据先被添加到缓存中。 此时有两种情况：

- state.length 为 0。 这时，在调用_read()前，state.needReadable 就会被设为 true。 因此，一定会调用 emitReadable()。 这个方法会在下一个 tick 中触发 readable 事件，同时再调用 flow()，从而形成流动。
- state.length 不为 0。由于流动模式下，每次都是从缓存中取第一个元素，所以这时 read()返回值一定不为 null。 故 flow()中的循环还在继续。

此外，从 push()的两个分支可以看出来，如果 state.flowing 设为 false，第一个分支便不会再进去，也就不会再调用 read(0)。 同时第二个分支中引发 flow 的调用后，也不会再调用 read()，这就完全暂停了底层数据的读取。

事实上，pause 方法就是这样使流从流动模式转换到暂停模式的。

背压反馈机制

考虑下面的例子：

```
const fs = require('fs')fs.createReadStream(file).on('data', doSomething)
```

监听 data 事件后文件中的内容便立即开始源源不断地传给 doSomething()。 如果 doSomething 处理数据较慢，就需要缓存来不及处理的数据 data，占用大量内存。

理想的情况是下游消耗一个数据，上游才生产一个新数据，这样整体的内存使用就能保持在一个水平。 Readable 提供 pipe 方法，用来实现这个功能。

pipe

用 pipe 方法连接上下游：

```
const fs = require('fs')fs.createReadStream(file).pipe(writable)
```

writable 是一个可写流 Writable 对象，上游调用其 write 方法将数据写入其中。 writable 内部维护了一个写队列，当这个队列长度达到某个阈值（state.highWaterMark）时， 执行 write()时返回 false，否则返回 true。

于是上游可以根据 write()的返回值在流动模式和暂停模式间切换：

```
readable.on('data', function (data) {
  if (false === writable.write(data)) {
    readable.pause()
  }
})
writable.on('drain', function () {
  readable.resume()
})
```

上面便是 pipe 方法的核心逻辑。

当 write()返回 false 时，调用 readable.pause()使上游进入暂停模式，不再触发 data 事件。 但是当 writable 将缓存清空时，会触发一个 drain 事件，再调用 readable.resume()使上游进入流动模式，继续触发 data 事件。

看一个例子：

```
const stream = require('stream')
var c = 0const readable = stream.Readable({
  highWaterMark: 2,
  read: function () {
    process.nextTick(() => {
      var data = c < 6 ? String.fromCharCode(c + 65) : null
      console.log('push', ++c, data)
      this.push(data)
    })
```

```
  }
})
const writable = stream.Writable({
  highWaterMark: 2,
  write: function (chunk, enc, next) {
    console.log('write', chunk)
  }
})
readable.pipe(writable)
```

输出：

```
push 1 A
write <Buffer 41>
push 2 B
push 3 C
push 4 D
```


虽然上游一共有 6 个数据（ABCDEF）可以生产，但实际只生产了 4 个（ABCD）。这是因为第一个数据（A）迟迟未能写完（未调用 next()），所以后面通过 write 方法添加进来的数据便被缓存起来。下游的缓存队列到达2时，write返回false，上游切换至暂停模式。 此时下游保存了 AB。 由于 Readable 总是缓存 state.highWaterMark 这么多的数据，所以上游保存了 CD。从而一共生产出来 ABCD 四个数据。

下面使用 tick-node 将 Readable 的 debug 信息按 tick 分组：

```
⌘ NODE_DEBUG=stream tick-node pipe.js
STREAM 18930: pipe count=1 opts=undefined
STREAM 18930: resume
```

```
---------- TICK 1 ----------

STREAM 18930: resume read 0

STREAM 18930: read 0

STREAM 18930: need readable false

STREAM 18930: length less than watermark true

STREAM 18930: do read

STREAM 18930: flow true

STREAM 18930: read undefined

STREAM 18930: need readable true

STREAM 18930: length less than watermark true

STREAM 18930: reading or ended false

---------- TICK 2 ----------

push 1 A

STREAM 18930: ondata

write <Buffer 41>

STREAM 18930: read 0

STREAM 18930: need readable true

STREAM 18930: length less than watermark true

STREAM 18930: do read

---------- TICK 3 ----------

push 2 B

STREAM 18930: ondata

STREAM 18930: call pause flowing=true

STREAM 18930: pause

STREAM 18930: read 0
```

```
STREAM 18930: need readable true

STREAM 18930: length less than watermark true

STREAM 18930: do read

---------- TICK 4 ----------

push 3 C

STREAM 18930: emitReadable false

STREAM 18930: emit readable

STREAM 18930: flow false

---------- TICK 5 ----------

STREAM 18930: maybeReadMore read 0

STREAM 18930: read 0

STREAM 18930: need readable false

STREAM 18930: length less than watermark true

STREAM 18930: do read

---------- TICK 6 ----------

push 4 D

---------- TICK 7 ----------
```


- TICK 0: `readable.resume()`
- TICK 1: `readable` 在流动模式下开始从底层读取数据
- TICK 2: `A` 被输出，同时执行 `readable.read(0)`。
- TICK 3: `B` 被输出，同时执行 `readable.read(0)`。 `writable.write('B')`返回 `false`。 执行 `readable.pause()`切换至暂停模式。
- TICK 4: TICK 3 中 `read(0)`引起 `push('C')`的调用，`C` 被加到 `readable` 缓存中。此时，`writable` 中有 `A` 和 `B`，`readable` 中有 `C`。 这时已在暂停模式，但在

readable.push('C')结束前，发现缓存中只有 1 个数据，小于设定的 highWaterMark（2），故准备在下一个 tick 再读一次数据。

- TICK 5: 调用 read(0)从底层取数据。
- TICK 6: push('D')，D 被加到 readable 缓存中。 此时，writable 中有 A 和 B， readable 中有 C 和 D。 readable 缓存中有 2 个数据，等于设定的 highWaterMark（2），不再从底层读取数据。

可以认为，随着下游缓存队列的增加，上游写数据时受到的阻力变大。 这种背压（back pressure）大到一定程度时上游便停止写，等到背压降低时再继续。

消耗驱动的数据生产

使用 pipe()时，数据的生产和消耗形成了一个闭环。 通过负反馈调节上游的数据生产节奏，事实上形成了一种所谓的拉式流（pull stream）。

用喝饮料来说明拉式流和普通流的区别的话，普通流就像是将杯子里的饮料往嘴里倾倒，动力来源于上游，数据是被推往下游的；拉式流则是用吸管去喝饮料，动力实际来源于下游，数据是被拉去下游的。

所以，使用拉式流时，是“按需生产”。 如果下游停止消耗，上游便会停止生产。所有缓存的数据量便是两者的阈值和。

当使用 Transform 作为下游时，尤其需要注意消耗。

```
const stream = require('stream')

var c = 0const readable = stream.Readable({

  highWaterMark: 2,

  read: function () {

    process.nextTick(() => {

      var data = c < 26 ? String.fromCharCode(c++ + 97) : null

      console.log('push', data)
```

```
      this.push(data)
    })
  }
})
const transform = stream.Transform({
  highWaterMark: 2,
  transform: function (buf, enc, next) {
    console.log('transform', buf)
    next(null, buf)
  }
})
readable.pipe(transform)
```

以上代码执行结果为：

```
push a
transform <Buffer 61>
push b
transform <Buffer 62>
push c
push d
push e
push f
```

可见，并没有将 26 个字母全生产出来。

Transform 中有两个缓存：可写端的缓存和可读端的缓存。

调用 transform.write()时，如果可读端缓存未满，数据会经过变换后加入到可读端的缓存中。 当可读端缓存到达阈值后，再调用 transform.write()则会将写操作缓存到可写端的缓存队列。 当可写端的缓存队列也到达阈值时，transform.write()返回 false，上游进入暂停模式，不再继续 transform.write()。

所以，上面的 transform 中实际存储了 4 个数据，ab 在可读端（经过了_transform 的处理），cd 在可写端（还未经过_transform 处理）。

此时，由前面一节的分析可知，readable 将缓存 ef，之后便不再生产数据。

这三个缓存加起来的长度恰好为 6，所以一共就生产了 6 个数据。

要想将 26 个数据全生产出来，有两种做法。 第一种是消耗 transform 中可读端的缓存，以拉动上游的生产：

```
readable.pipe(transform).pipe(process.stdout)
```

第二种是，不要将数据存入可读端中，这样可读端的缓存便会一直处于数据不足状态，上游便会源源不断地生产数据：

```
const transform = stream.Transform({
  highWaterMark: 2,
  transform: function (buf, enc, next) {
    next()
  }
})
```